Leveraging Register Windows to Reduce Physical Registers to the Bare Minimum

Eduardo Quiñones, *Member*, *IEEE*, Joan-Manuel Parcerisa, and Antonio González, *Member*, *IEEE*

Abstract—Register window is an architectural technique that reduces memory operations required to save and restore registers across procedure calls. Its effectiveness depends on the size of the register file. Such register requirements are normally increased for out-of-order execution because it requires registers for the in-flight instructions, in addition to the architectural ones. However, a large register file has an important cost in terms of area and power and may even affect the cycle time. In this paper, we propose a software/hardware early register release technique that leverage register windows to drastically reduce the register requirements, and hence, reduce the register file cost. Contrary to the common belief that out-of-order processors with register windows would need a large physical register file, this paper shows that the physical register file size may be reduced to the bare minimum by using this novel microarchitecture. Moreover, our proposal has much lower hardware complexity than previous approaches, and requires minimal changes to a conventional register window scheme. Performance studies show that the proposed technique can reduce the number of physical registers to the number of logical registers plus one (minimum number to guarantee forward progress) and still achieve almost the same performance as an unbounded register file.

Index Terms—Register windows, physical register file, early register release.

1 Introduction

Register window is an architectural technique that reduces the amount of loads and stores required to save and restore registers across procedure calls by storing the local variables of multiple procedure contexts in a large architectural register file. When a procedure is called, it maps its context to a new set of architected registers, called a register window. Through a simple runtime mechanism, the compiler-defined local variables are then renamed to these windowed registers.

If there are not enough architectural registers to allocate all local variables, some local variables from caller procedures are saved to memory and their associated registers are freed for the new context. When the saved variables are needed, they are restored to the register file. These operations are typically referred to as *spill* and *fill*. SPARC [6] and Itanium [10] are two commercial architectures that use register windows.

The effectiveness of the register windows technique depends on the size of the architectural register file because the more registers it has, the less spills and fills are required [28]. Besides, for an out-of-order processor, the number of architectural registers determines the size of the rename map table, which, in turn, determines the minimum number of physical registers.

To extract high levels of parallelism, out-of-order processors use many more physical registers than architected ones, to store the uncommitted values of a large number of instructions in flight [8]. Therefore, an out-of-order processor with register windows requires a large amount of physical registers because of a twofold reason: to hold multiple contexts and support a large instruction window. Unfortunately, the size of the register file has a strong impact on its access time [8], which may stay in the critical path that sets the cycle time. It has also an important cost in terms of area and power [31]. There exist many proposals that address this problem through different approaches.

One approach consists of pipelining the register file access [11]. However, a multicycle register file requires a complex multiple-level bypassing and increases the branch misprediction penalty. Other approaches improve the register file access time, area, and power by modifying the internal organization, through register caching [30] or register banking [5]. Alternative approaches have focused on reducing the physical register file size by reducing the register requirements through more aggressive reservation policies: late allocation [9], [19], [23] and early release [3], [14], [17], [18], [20], [22].

In this paper, we propose a new software/hardware early register release technique for out-of-order processors that builds upon register windows to achieve an impressive level of register savings. On conventional processors, a physical register remains allocated until the next instruction writing the same architectural register commits. However, procedure call and return semantics enables more aggressive conditions for register release:

1. When a procedure finishes and its closing return instruction commits, all physical registers defined by this procedure can be safely released. The values defined in the closed context are *dead values* that will never be used again.
2. When the rename stage runs out of physical registers, mappings defined by instructions that are

- *E. Quiñones is with the Barcelona Supercomputing Center, Spain. E-mail: eduardo.quinones@bsc.es.*
- *J.-M. Parcerisa and A. González are with the Computer Architecture Department, Politechnical University of Catalonia, Spain. E-mail: jmanel@ac.upc.edu, antonio.gonzalez@intel.com.*

Manuscript received 26 Mar. 2008; revised 19 Jan. 2010; accepted 26 Jan. 2010; published online 9 Apr. 2010.
Recommended for acceptance by K. Ghose.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-03-0129.
Digital Object Identifier no. 10.1109/TC.2010.85.

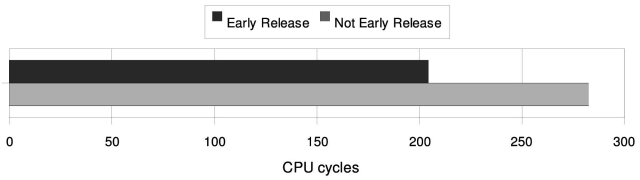

Fig. 1. Average lifetime of physical registers for the set of integer benchmarks executing in a processor with an unbounded register file, when applying our early release techniques and when not applying them.

not in-flight, which belong to caller procedures, can also be released. However, unlike the previous case, these values may be used in the future, after returning from the current procedure, so they must be saved to memory before they are released.

3. Architectural register requirements vary along a program execution. The compiler can compute the register requirements for each particular control flow path and insert instructions to enlarge or shrink the context depending on which control flow path is being executed. When a context is shrunk, all defined physical registers that lay outside of the new context contain dead values and can be safely released.

By exploiting these software/hardware early release opportunities, the proposed scheme achieves a drastic reduction in physical register requirements. By applying them to a processor with an unbounded physical register file, the average register *lifetime* (number of cycles between the allocation and release of a physical register) drops by 30 percent (see Fig. 1). This allows to reduce the number of physical registers to the minimum number that still guarantees forward progress, i.e., same number of architectural plus one (128 in our experiments for IPF binaries), and still achieve almost the same performance as an unbounded register file.

Contrary to the common belief that out-of-order processors with register windows would need a large physical register file, this paper shows that register windows, together with the proposed technique, can significantly reduce the physical register file pressure to the bare minimum. In other words, we show that the proposed scheme achieves a synergistic effect between register windows and out-of-order execution, resulting in an extremely cost-effective implementation.

Besides, our scheme requires much lower hardware complexity than previous related approaches [22], and it requires minimal changes to a conventional register windows scheme.

As stated above, a register windows mechanism works by translating compiler-defined local variables to architected registers prior to renaming them to physical registers. The information required for this translation is kept as a part of the processor state and must be recovered in case of branch mispredictions or exceptions. Our scheme also provides an effective recovery mechanism that is suitable for out-of-order execution. This paper extends the work in [26] in several ways:

- It proposes a new early register release compiler technique.
- It studies the effect of reducing the maximum register window size.
- It analyzes the average lifetime of physical registers when applying our proposals.
- It introduces the *Retirement Map Table* that allows releasing those physical registers whose architectural register has been already redefined at the rename stage.

The rest of this paper is organized as follows: Section 2 discusses the state of the art on early register release techniques. Section 3 describes our proposal in detail. Section 4 presents and discusses the experimental results. Section 5 analyzes several cost and complexity issues of the proposed solution and previous approaches. Finally, the main conclusions are summarized in Section 6.

2 Related Work

In this section, we will shortly review techniques that make early releasing of physical registers, to reduce the average number of required registers.

Jones et al. [13], [14] use the compiler to identify registers that will be read a given number of times (n) known at compile time, and rename them to different logical registers. Upon issuing an instruction with one of these logical registers as a source, the processor can release the register through checkpointing if it is the nth read operation of this register.

Moudgill et al. [21] suggested releasing physical registers eagerly, as soon as the last instruction that uses a physical register commits. The last-use tracking is based on counters which record the number of pending reads for every physical register. This initial proposal support precise exceptions by adding an *unmapped* flag to each physical register, which is set when a subsequent instruction redefines the logical register it has associated. Then, a physical register can be released once its usage counter is zero and its *unmapped* flag is set. More recently, Akkary et al. [3] proposed to improve the Moudgill scheme by using a map table checkpoints. For proper exception recovery of the reference counters, when a checkpoint is created, the counters of all physical registers belonging to the checkpoint are incremented. Similarly, when a checkpoint is released, the counters of all physical registers belonging to the checkpoint are decremented.

Martin et al. [17] introduced the *Dead Value Information* (DVI), which is calculated by the compiler and passed to the processor to help with early register releasing. DVI can be passed through explicit ISA extensions instructions, which

contain a bit-mask indicating the registers to release, or implicitly on certain instructions such as procedure call and returns. They use the DVI such that when a dynamic call or return is committed, the caller-saved registers are early-released because the calling conventions implicitly state that they will not be live. A similar approach was introduced by Lo et al. [16] but applied to simultaneous multithreading processors. Although similar to our approach, both approaches require introducing new ISA instructions.

Monreal et al. [20] proposed a scheme where registers are released as soon as the processor knows that there will be no further use of them. Conventional renaming forces a physical register to be idle from the commit of its *Last-Use* (LU) instruction until the commit of the first *Next-Version* (NV) instruction. The idea is to shift the responsibility from NV instruction to LU instruction. Each time a NV instruction is renamed, its corresponding LU instruction pair is marked. Marked LU instructions reaching the commit stage will release registers instead of keeping them idle until the commit of the NV instruction.

Ergin et al. [7] introduced the checkpointed register file to implement early register release. This scheme can release a physical register before the redefining instruction is known to be nonspeculative. This is done by copying its value into the shadow bit cells of the register where it can be accessed easily if a branch misprediction occurs.

Martinez et al. [18] presented *Cherry*: Checkpointer Early Resource Recycling, a hybrid mode of execution based on reorder buffer and checkpointing that decouples resource recycling, i.e., resource releasing, and instruction retirement. In this scheme, physical registers are recycled if both the instructions that produce the value and all their consumers have been executed, which are identified by using the scheme described in [21], and are free of replay traps and are not subject to branch mispredictions. To reconstruct the precise state in case of exceptions or interrupts, the scheme relies on periodic register file checkpointing.

Balkan et al. [4] proposed to early deallocate a written physical register and to reallocate it again to a subsequent instruction being renamed. In case not all consumers have issued at that time, the reallocating instruction is either stalled at dispatch, or it speculatively proceeds forward and rechecks this condition again just before writeback. In case the condition fails again, it is either provided a new free register or reissued. However, stalling the reallocating instructions can significantly impact on the performance and providing free registers for the reallocating instructions at writeback requires significant hardware complexity.

Oehmke et al. [22] have recently proposed the *virtual context architecture* (VCA), which maps logical registers holding local variables to a large memory address space and manages the physical register file as a cache that keeps the most recently used values. Logical register identifiers are converted to memory addresses, and then, mapped to physical registers by using a tagged set-associative rename map table. Unlike the conventional renaming approach, the VCA rename table lookup may miss, i.e., there may be no physical register mapped to a given logical register. When the renaming of a source register causes a table miss, the value is restored from memory and a free physical register is allocated and mapped onto the table. If there are no free physical registers or table entries for a new mapping, then a replacement occurs: a valid entry is chosen by LRU, the value is saved to memory and its physical register is released. Although the VCA scheme does not properly define register windows as such, in practice, it produces similar effects: multiple procedure contexts are maintained in registers, and the available register space is transparently managed without explicit saves and restores. This also allows VCA being applied to Simultaneous Multithreading processors. However, unlike our approach, their tagged set-associative map table adds substantial complexity to the rename stage, which might have implications on the cycle time. Furthermore, when a map table entry is replaced, its associated value is always saved to memory, regardless of whether it is actually a *dead value*. This occurs because VCA is not able to distinguish *dead values* from *live values*.

Some current commercial architectures implement register windows to reduce procedure call/return overhead: SPARC [6] and Itanium [10]. In the former case, where register windows are of fixed size, overflows and underflows are handled by trapping to the operating system. In the latter case, where register windows are of variable size, overflows and underflows are solved by a hardware mechanism called *Register Stack Engine* that transparently handles saves and restores through procedure calls.

3 Early Register Release with Register Windows

This section describes our early register release techniques based on register windows. Our scheme assumes an ISA with full register window support, such as IA-64 [12]. Along this paper, we assume a conventional out-of-order processor with a typical register renaming mechanism that uses a map table to associate architected to physical registers. The IA-64 defines 32 static registers and 96 windowed registers. The static registers are available to all procedures, while the windowed registers are allocated on demand to each procedure. Both static and windowed architected registers map to a unified physical register file. Our technique applies to windowed registers.

In the following sections, a basic register window mechanism for out-of-order processors is explained first. Then, we expose the early register release opportunities and present a mechanism to exploit them.

3.1 Baseline Register Window

Register window is a technique that helps to reduce the loads and stores required to save registers across procedure calls by storing the local variables of multiple procedure contexts in a large register file [28]. Throughout this paper, we will use also the term procedure context to refer to a register window.

From the ISA's perspective, all procedure contexts use the same "virtual" register name space. However, when a procedure is called, it is responsible for dynamically allocating a separate set of consecutive architected registers, a register window, by specifying a *context base pointer* and a window *size*. Each virtual register name is then dynamically translated to an architected register by simply adding the

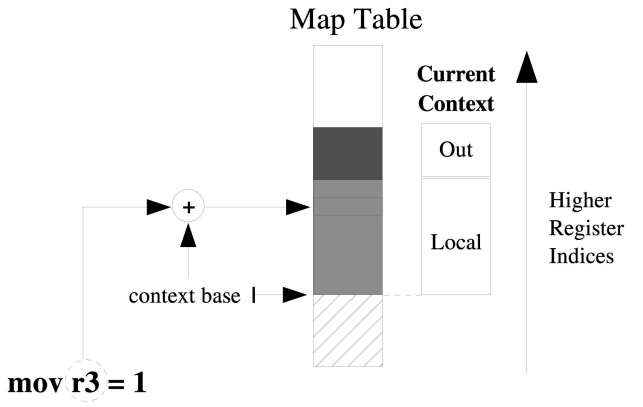

Fig. 2. Dynamic translation from virtual register *r3* to its corresponding architectural windowed register.

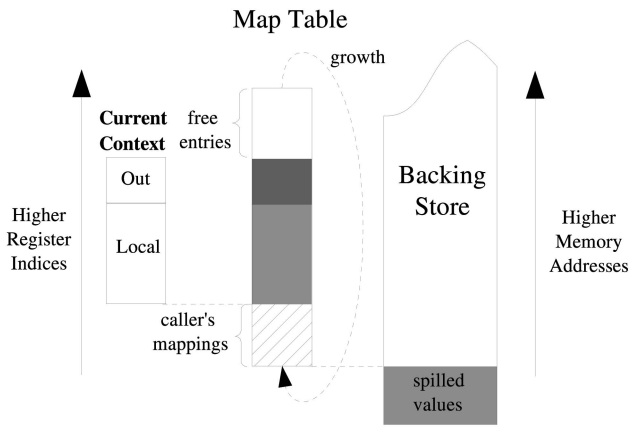

Fig. 4. Relationship between map table and backing store memory.

base pointer to it (see Fig. 2). Note that register windows grow toward higher register indexes.

Every register window is divided into two regions: the *local region*, which includes both input parameters and local variables, and the *out region*, where it passes parameters to its callee procedures. By overlapping register windows, parameters are passed through procedures, so these registers holding the output parameters of the caller procedure become the local parameters of the callee. The overlap is illustrated in Fig. 3.

Hence, every register window is fully defined by a *context descriptor* having three parameters: the *context base pointer*, which sets the beginning of the register window; the *context size*, which includes the local and out regions; and the *output parameters*, which defines the size of the out region.

Register windows are managed by software, with three specific instructions: *br.call, alloc*, and *br.ret*. Br.call branches to a procedure and creates a new context, by setting a context descriptor with its base pointing to the first register in the out region of the caller context, and its context size equal to the out region's size. Alloc modifies the current context descriptor by setting a new context size and output parameters based on the information coded inside the instruction. This instruction is commonly executed just after the br.call to enlarge the register window, defining the size of the local and out regions of the new context. However, note that the new size may also be smaller than the previous size, so the alloc may be used either for enlarging or shrinking the context. Finally, br.ret returns to the caller procedure and makes its context the current context. The compiler is responsible for saving, on each procedure call, the caller context descriptor to a local register and restoring it later on return.

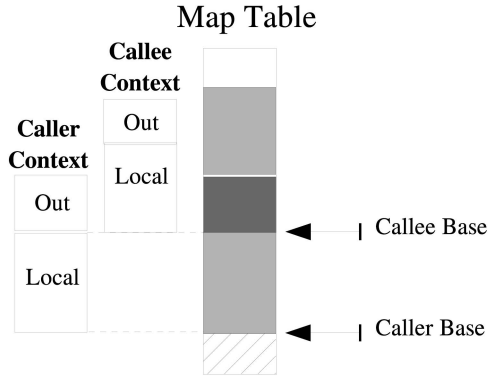

Fig. 3. Overlapping register windows.

It may happen that an alloc instruction increases the window size but there is no room available to allocate the new context size in the architectural register file, i.e., at the top of the map table. In such case, the contents of some mappings and their associatived physical register at the bottom of the map table (which belong to caller procedure contexts) are sent to a special region of memory called *backing store*, and then, are released. Such operation is called a *spill*. Note that spills produce both free physical registers and free entries in the map table. These entries can then be assigned to new contexts to create the illusion of an infinite-sized register stack.[1]

When a procedure returns, it expects to find its context in the map table and the corresponding values stored in physical registers. However, if some of these registers were previously spilled to memory, the context is not completely present. For each missing context register, a physical register is allocated and renamed in the map table. Then, the value it had before spilling is reloaded from the backing store. Such operation is called a *fill*. Subsequent instructions that use this value are made data dependent on the result of the fill, so they are not stalled at renaming to wait for the fill completion.

The relationship between map table and backing store is shown in Fig. 4. Note that the map table is managed as a circular buffer. Spill and fill operations will be discussed in more detail in Section 3.6.

3.2 Early Register Release Techniques with Compiler Support

On a conventional out-of-order processor, a physical register is released when the instruction that redefines it commits. To help reduce the physical register pressure, we have observed that by shrinking the context size, mappings that lay above the new shrunk context can be early released. We have identified an early release opportunity: the *Alloc Release*.

1. Each mapping has a unique associated backing store address.

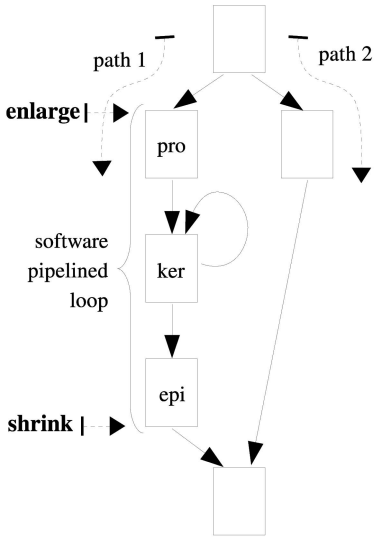

Fig. 5. Architectural register requirements are higher in *path 1* than *path 2*.

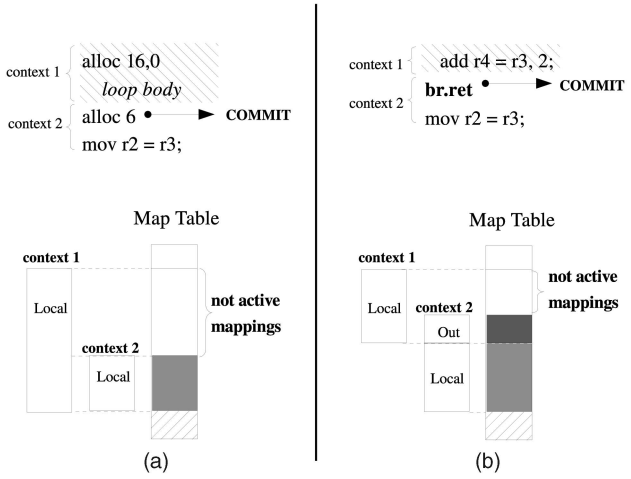

Fig. 6. In both figures, context 1 mappings that do not overlap with context 2 are *not-active* (none in-flight instructions refer them), so they can be released. (a) *Alloc-Release* technique. (b) *Context-Release* technique.

Alloc Release. Architectural register requirements may vary along the control flow graph. At compile time, if a control flow path requires more registers, the context can be enlarged by defining a higher context size with an alloc instruction. At runtime, new architectural registers are reserved at rename stage, as explained in Section 3.1. Analogously, if the architectural register requirements decrease, the compiler can shrunk the context by defining a lower context size with an alloc instruction. At runtime, when the alloc instruction commits, none of the architectural registers that lay above the shrunk context are referenced by any of the currently in-flight instructions. We refer to them as *not-active* mappings, and they can be early released as well as their associated physical registers without having to wait until the commit of a subsequent redefinition. In both cases (to enlarge or shrink the context size), the compiler needs to introduce an alloc instruction. Fig. 5 shows a control flow graph with unbalanced architectural register requirements. Alloc Release is illustrated in Fig. 6a.

Compiler support may help to reduce the physical register pressure by shrinking contexts and releasing not-active mappings that lay above the shrunk context. In fact, what this technique actually does is to adapt better the architectural registers requirements. However, physical register requirements are not available at compile time. To overcome this limitation, we propose two hardware techniques called *Context Release* and *Register Release Spill*.

3.3 Early Register Release Techniques without Compiler Support

Context Release takes full advantage of call conventions. When a procedure finishes and its closing br.ret instruction commits, many of the physical registers defined inside the closed procedure become not-active and can be early released. Context Release is illustrated in Fig. 6b. Note that this technique is very similar to Alloc Release, since br.ret also shrinks the current context size. However, although both techniques require the same hardware support, the compiler is not aware of applying Context Release.

Register Release Spill. This technique is based on an extension of the not-active mapping concept: if none of the instructions of a procedure are currently in-flight, not all mappings of the procedure context need to stay in the map table. Hence, not-active mappings are not only those mappings that lay above a shrunk context, but also those mappings that belong to previous caller procedure contexts not currently present in the processor. The Register Release Spill technique is illustrated in Fig. 7. As shown in Fig. 7a, when *br.call* commits, context 1 mappings become not-active. This is not the case in Fig. 7b, where the context 1 procedure has already returned before br.call commits, so their mappings have become active. Hence, only not-active mappings in Fig. 7a can be early released. However, since these mappings belong to caller procedures, they contain live values and must first spill the content of their associated physical registers before releasing them. We will refer to this operation as Register Release Spill.

Though beneficial for register pressure, this technique increases the amount of spill operations. Fig. 8 shows the number of spill operations issued with respect to the total number of committed instructions when applying our

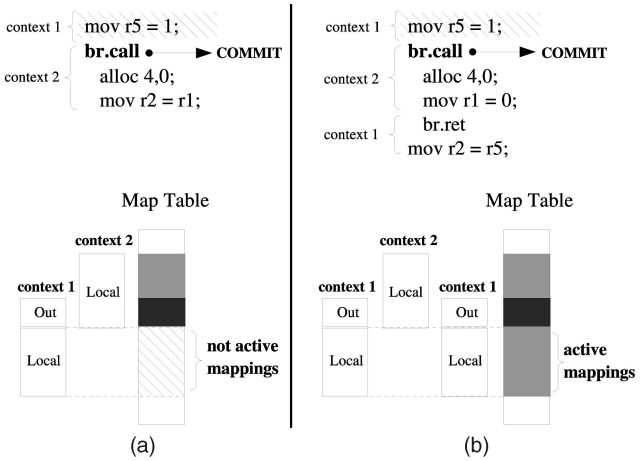

Fig. 7. *Register-Release Spill* technique. (a) Context 1 mappings are *not-active*, so they can be early released. (b) Context 1 mappings are *active* (after returning from the procedure), so they cannot be released.

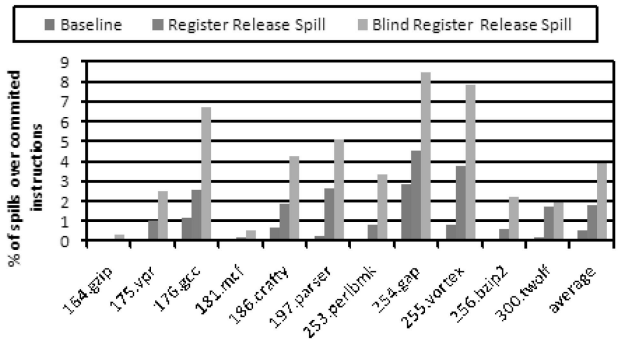

Fig. 8. Number of spill operations issued with respect to the total number of committed instructions when applying our baseline scheme (*Baseline*), a blind application of the Register Release Spill (*Blind Register Release Spill*), and applying Register Release Spill only when the renaming runs out of physical registers.

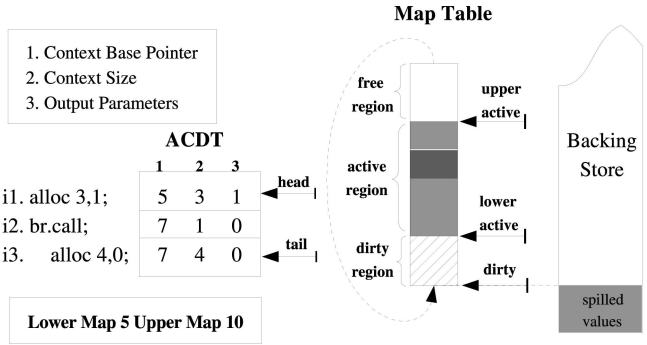

Fig. 9. The pointers *upper active*, *lower active*, and *dirty* divide the map table in three regions: *free*, *active*, and *dirty*. *Upper active* and *lower active* pointers are computed using ACDT information. *Dirty* pointer points to the first nonspilled mapping.

baseline scheme and a blind application of the Register Release Spill (labeled as *Baseline* and *Blind Register Release Spill*, respectively). On average, a blind application of the Register Release Spill increases the amount of spill operations from 0.5 to four percent over the total number of committed instructions. Since spilling registers has an associated cost, it could reduce the benefits brought by register windows. Hence, Register Release Spill is only triggered when the released registers are actually required for the execution of the current context, i.e., if the renaming runs out of physical registers. As shown in Fig. 8, this policy (labeled as *Register Release Spill*) produces, on average, only 1.8 percent spill operations over the total number of committed instructions (see Section 4.1 for further details).

Note that there is a slight difference between the Register Release Spill and the conventional spill used in the baseline (see Section 3.1). A conventional spill is triggered by a lack of mappings when the window size is enlarged. What Register Release spill actually does is "to steal" physical registers from caller procedures at runtime to ensure an optimum execution of the current active zone.

3.4 Implementation

To implement register windows in an out-of-order processor, we propose the *Active Context Descriptor Table* (ACDT). The ACDT tracks all uncommitted context states to accomplish two main purposes: to identify the active mappings at any one point in time, which is required by our early register release techniques; and to allow precise state recovery of context descriptor information in case of branch mispredictions and other exceptions. Moreover, we introduce the *Retirement Map Table* that holds the rename map table at commit stage. It allows that Alloc Release and Context Release techniques identify at commit stage the not-active mappings that can be released, even when these mappings have been already redefined at rename stage by another procedure context.

ACDT. The ACDT acts as a checkpoint repository that buffers in a *fifo* way, the successive states of the current context descriptor. As such, a new entry is queued for each context modification (at rename stage in program order), and it is removed when the instruction that has inserted the checkpoint is committed. Hence, ACDT maintains only active context descriptors. In case the ACDT is full at the time a new entry is going to be queued, the rename stage is stalled until the instruction that has insert the oldest active context commits. With this information, the ACDT allows precise state recovery of context descriptor information: in case of branch mispredictions and other exceptions, when all younger instructions are squashed, the subsequent context states are removed from the ACDT too.

As they are defined, active mappings stay in a continuous region on the map table, called *active region*. The active region is bounded by two pointers that are calculated using the ACDT information: the *lower active* and the *upper active*. Both pointers are updated at each context modification using the information present in the ACDT: the *lower active* equals to the minimum context base pointer present in the ACDT, and the *upper active* equals to the maximum sum of base + size present in the ACDT.

The mappings staying below the *lower active* pointer belong to not-active contexts of callers procedures. These mappings may eventually become free if their values are spilled to memory, or may become active again if a br.ret restores that context. They form the so-called *dirty region* of the map table, and it lays between the *lower active* pointer and a third pointer called *dirty*. The *dirty* pointer equals to the first mapping that will be spilled if the renaming engine requires it. Actually, the associated backing store address of the *dirty* pointer corresponds to the top of the backing store stack.

Finally, the *free-region* lays between the *upper active* pointer and the *dirty* pointer (note that the map table is managed as a circular buffer), and it contains invalid mappings.

The three regions are shown in Fig. 9. When the first instruction (*i1*) is renamed, the lower active pointer is set to five and the upper active pointer is set to 7 since the new context size is set to three (field 2 of the ACDT). When *i2* is renamed, the current context base is set to the out region of the previous context, i.e., seven. However, at this time, pointers are not updated since the new context has still not enlarged its context size. This is done at the time *i3* is renamed, in which the new size is set to four, increasing the upper active context up to 10. Thus, after *i3*, there are three active contexts with a total size of six architectural registers.

Retirement Map Table. When a context is shrunk, mappings that become not-active are not always accessible. As shown in Fig. 10, when alloc *i2* is renamed, it creates the

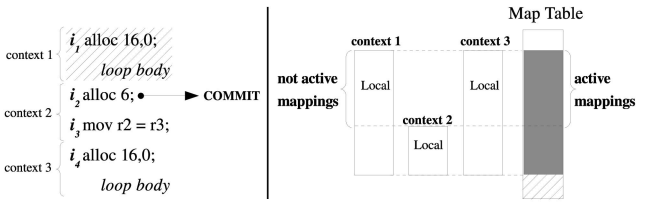

Fig. 10. Not-active mappings from *context 1* have become active because of *context 3*, so they cannot be released if *context 2* commits.

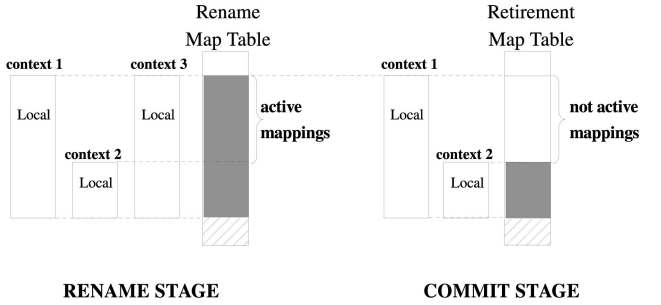

Fig. 11. The *Retirement Map Table* holds the committed state of the *Rename Map Table*. *Context 1* mappings will remain into the Retirement Map Table until *Context 3* commits.

descriptor *context 2* and shrinks the context, so the deallocated mappings of *context 1* become not-active. Next, when alloc *i4* is renamed, it creates a new descriptor *context 3*, increases the size of the context again, and reuses the map entries that were released by alloc *i2*. At that point, if the alloc *i2* commits, the mappings of *context 1* cannot be released because their corresponding map entries have been reassigned to *context 3*, which is active. However, the physical registers associated to *context 1* can be early released because they contain dead values.

In order to obtain *context 1* defined physical registers, we introduce the *Retirement Map Table* [11], [21] that holds the state of the *Rename Map Table* at commit stage. When an instruction commits, the Retirement Map Table is indexed using the same Rename Map Table register index, and updated with its corresponding destination physical register (which indicates that the physical register contains a committed value). When a branch misprediction or exception occurs, the offending instruction is flagged, and the recovery actions are delayed until this instruction is about to commit. At this point, the Retirement Map Table contains the architectural state just before the exception occurred. Then, the processor state is restored by copying the Retirement Map Table to the Rename Map Table (to avoid significant penalty stalls caused by the use of Retirement Map Table, some optimizations have been proposed [3]).

Fig. 11 shows the same previous example but introducing the Retirement Map Table. Note that when the alloc that has created the context descriptor *context 2* commits, not-active mappings from *context 1* are still present in the Retirement Map Table, so they can be safely early released.

In summary, when a shrunk context commits, Alloc Release and Context Release techniques are applied by releasing those not-active mappings from the Retirement Map Table that have laid above the committed shrunk context. Furthermore, when the rename stage runs out of physical registers, a Register Release Spill operation is triggered and releases as many not-active mappings as needed from the *dirty region* starting at the *dirty* pointer. These spilled mappings become part of the *free region*.

3.5 Delayed Spill/Fill Operations

A closer look to the register window scheme for out-of-order processors described in the previous sections reveals that there is still some room for physical register pressure reduction.

First, when a br.ret instruction is executed and the restored context is not present in the map table, fill operations are generated until all its mappings are restored. Assuming a realistic scenario, there may exist a limit on the number of fills generated per cycle. So, a massive fill generation may stall the renaming for several cycles. Moreover, it may happen that some of these mappings will never be used. Actually, it occurs quite often that a br.ret is followed by another br.ret, so none of the restored mappings are used. Hence, unnecessary fill operations reserve unnecessary physical registers, which means a higher register file pressure. Hence, we propose to defer each fill and its corresponding physical register allocation until the mapping is actually used by a subsequent instruction. By delaying fill operations, we achieve a lower memory traffic and reduce the physical register pressure which results in a higher performance.

Second, when an alloc is executed and there is not enough space into the map table, spill operations are generated to free the required mappings. As it happens with the previous case, a massive spill generation may stall the renaming for several cycles. Hence, we propose to defer each spill until the map table entry is actually reassigned by a subsequent instruction.

Hence, when a br.ret is executed and the restored context is not present in the map table, the required mappings from the free region are appended to the active region and marked with a *pending fill* bit, so the br.ret instruction is not stalled. When a subsequent dependent instruction wants to use a mapping with the pending fill bit set, a fill operation is generated and a new physical register is reserved. In a similar way, when an alloc is executed and there is not enough space for the new context, the required mappings from the dirty region are appended to the active region and marked with a *pending spill* bit, so the alloc is not stalled. When an subsequent instruction redefines a mapping with its pending spill bit set, the current mapping is first spilled to the backing store.

It might happen that a mapping with the pending spill bit set is not redefined until a subsequent nested procedure, and the map table has wrapped around several times. In that case, it would be costly to determine its associated backing store address. Thus, it is less complex to have an engine that autonomously clears the pending spills that were left behind. This problem does not occur with pending fills because these are actually invalid mappings that may be safely reused.

3.6 Spill-Fill Operations

This section discusses some implementation considerations about the spill and fill operations.

A naive implementation of spills and fills could insert them into the pipeline as standard store and load instructions. However, spills and fills have simpler requirements that enable a more efficient implementation. First, their effective addresses are known at rename time and the data that the spills store to memory are committed values. Hence, all their source operands are ready at renaming, which allows using a much simpler hardware scheduler. Second, our system guarantees that spills and fills do not require memory disambiguation with respect to program stores and loads as the stack address space is not accessible by regular loads/stores. Thus, spill and fill operations can be scheduled independently from all other program instructions by using a simple *fifo* buffer [22]. To that end, we introduce a dedicated Spill/Fill issue queue (see Section 4.1 for further details).

4 Evaluation

This section evaluates the proposed early register release techniques (Alloc Release, Context Release, Register Release Spill, and Delayed Spill/Fill) presented in previous sections, and compares them to a configuration that uses the VCA register windows scheme (see Section 2). For each configuration, performance speedups are normalized IPCs relative to a configuration that uses the baseline register window scheme with 160 physical registers. Although this gives an advantage of 32 registers to the baseline, we believe that it is a more reasonable design point for such an out-of-order processor with 128 logical registers. We will show that although our best scheme has much less registers, not only it outperforms the 160-registers baseline, but it still performs within a 192-registers baseline. At the end of this section, we provide a more complete performance comparison for a wide range of physical register file sizes.

4.1 Experimental Setup

All the experiments presented in this paper use a cycle-accurate, execution-driven simulator that runs IA-64 ISA binaries. It was built from scratch using the Liberty Simulation Environment (LSE) [29]. LSE is a simulator construction system, based on module definitions and module communications, that also provides a complete IA-64 functional emulator that maintains the correct machine state.

We have simulated the integer benchmark SpecCPU2000 suite [2] (except two integer benchmarks (*eon, vortex*), which the LSE emulator environment could not execute) using the Minne Spec [15] input set. Floating-point benchmarks have not been evaluated because the IA-64 register window mechanism is only implemented on integer registers. All benchmarks have been compiled with IA-64 Opencc [1] compiler using maximum optimization levels. For each benchmark, 100 million committed instructions are simulated. To obtain representative portions of code to simulate, we have used the Pinpoint tool [24].

An eight-stage out-of-order processor is fully simulated. We have paid special attention to the rename stage implementation, modeling many IA-64 peculiarities that are involved in the renaming, such as the register rotation and the application registers. The register stack engine has been replaced by our scheme. The microarchitecture features predicate prediction [25], [27]. Load-store queues, as well as the data and control speculation mechanisms defined in IA-64, are also modeled and integrated in the memory disambiguation subsystem. The main architectural parameters are shown in Table 1.

TABLE 1
Main Architectural Parameters Used

Architectural Parameters	
Fetch Width	Up to 2 bundles (6 instructions)
Issue Queues	Integer Issue Queue: 80 entries Floating-point Issue Queue: 80 entries Branch Issue Queue: 32 entries Load-Store Queues: 64 entries each
Reorder Buffer	256 entries
L1D	64KB, 4way, 64B block, 2 cycle latency Non-blocking, 12 primary misses, 4 secondary misses, 16 write-buffer entries 2 load, 2 store ports
L1I	32KB, 4 way, 64B block, 1 cycle latency
L2 unified	1MB, 16 way, 128B block, 8 cycle latency Non-blocking, 12 primary misses 8 write-buffer entries
DTLB ITLB	512 entries. 10 cycles miss penalty 512 entries. 10 cycles miss penalty
Main Memory	120 cycles of latency
Fetch Branch Predictor	Gshare 14-bit GHR, 4KB, 1-cycle acces
Predicate Predictor	Perceptron. 30b GHR. 10b LHR Total size :148 KB. 3-cycle access 10 cycles for misprediction recovery
Integer Map Table	96 local entries, 32 global entries
Integer Physical Register File	129 physical registers

The simulator models in detail the baseline register windows, as well as our proposed techniques (Alloc Release, Context Release, Register Release Spill, and Delayed Spill/Fill), the ACDT mechanism, and the Retirement Map table, as described in the previous section. In order to manage the spill/fill operations, the simulator introduces a dedicated 16-entry spill/fill issue queue as described in the previous section. Moreover, the simulator also models the VCA register windows technique, featuring a four-way set associative cache for logical-physical register mapping, with 11-bit tags, as described in [22], although it was adapted to the IA-64 ISA.

4.2 Estimating the Potential of the Alloc Release

The Alloc Release technique releases not required architectural registers (and their associated physical registers) by introducing alloc instructions that shrink the context size (see Section 3.2). A context can be shrunk by deallocating the mappings that are above the highest architectural register holding a live value. Hence, the compiler should assign the shortest lifetime values to the highest architectural register indexes in order to release them as soon as possible by shrinking its context. However, this register assignment is not trivial, since the same value may have different lifetimes depending on the control flow path. Moreover, the introduction of shrunk allocs increases the program code size, having undesirable side effects on the instruction cache and the fetch bandwidth. Fig. 12 shows the number of times there is one, eight, or 16 architectural registers that can be released at the beginning of each basic block if the context is perfectly shrunk (i.e., shortest lifetimes are allocated into the highest context indexes), when executing 100 million of committed instructions.

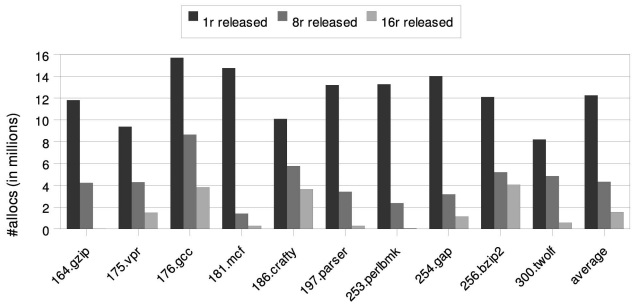

Fig. 12. Number of allocs that the compiler should introduce when executing 100 million committed instructions if a perfect shrunk of contexts is performed.

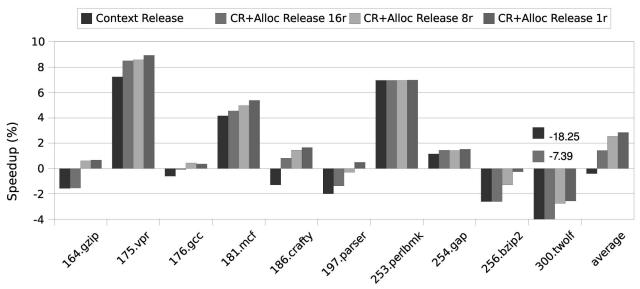

Fig. 13. Performance evaluation of Context Release and several configurations with Alloc Release and Context Release. Speedups are normalized to the baseline register window scheme with 160 physical registers.

Evaluating the Alloc Release technique requires to compare optimized and unoptimized binaries. For a fair comparison, both binaries should be run until completion, not just running a portion of the execution, because code optimized with Alloc Release is different from a code that does not apply it. Because of the limited resources of our experimental infrastructure, we took an alternative workaround: we attempted to estimate the *potential* performance speedup of this technique, by evaluating an upper bound of the Alloc Release.

Instead of inserting alloc instructions, the compiler annotates the code at the beginning of each basic block with a list of *live* architectural registers at that point. A register is considered *live* in a given point if there is at least one consumer in any possible path downstream. At runtime, when the first instruction of a basic block commits, the simulator releases the physical registers associated to current context mappings not included in the live register list, as would have occurred with a shrinking alloc instruction. The physical registers are easily identified from the Retirement Map Table.

However, note that by generating simple live register lists, there is no guarantee that the freed registers are located at the highest mappings of the context, to make the context shrink possible. Thus, the upper bound configuration does not attempt to shrink the context but only to free the physical registers that correspond to nonlive values. This is the effect that would have been seen on the Alloc Release if the nonlive registers are always mapped by the compiler to the highest context positions, which is obviously an optimistic assumption.

4.3 Performance Evaluation of Our Early Register Release Techniques

This section compares in terms of performance our proposals: Alloc Release, Context Release, Register Release Spill, and Delayed Spill/Fill. Each of them, either can be implemented independently, or as a combination of one with the rest. Although a thorough study has been realized, only the best four configurations are presented:

1. Context Release,
2. Alloc Release and Context Release,
3. Register Release Spill and Context Release, and
4. Delayed Spill/Fill and Register Release Spill and Context Release.

Note that the four configurations use Context Release. This is because Context Release is a very powerful technique that does not require any binary code modification and does not carry any side effect, unlike Alloc Release or Register Release Spill. Moreover, it is a low-cost technique in terms of hardware.

Alloc Release and Context Release. Fig. 13 compares configurations 1 and 2, presented in the previous paragraph, in terms of performance. The first column uses only Context Release technique. The remaining three columns are variants of Context Release and Alloc Release configurations, where Alloc Release is applied only if the number of released physical registers is bigger than 16, eight, or one, respectively. On average, applying a nonrestricted Alloc Release technique, i.e., release a physical register when possible, together with Context Release (fourth column) achieves a performance advantage of more than three percent. However, such a performance improvement does not make up for the amount of alloc instructions that the compile should insert, as shown in Fig. 12. The restrictive use of Alloc Release (second and third columns) can drastically reduce the code size expansion (from 12 to two and four millions of generated allocs), but it carries a performance loss of 1.5 and 0.4 percent, respectively, in comparison to a nonrestrictive use of Alloc Release.

One could expect a higher performance improvement, but the Alloc Release technique shrinks contexts because the architectural register requirements decreases, and not because of a lack of physical registers. However, there is one benchmark (*twolf*) that achieves a performance advantage of 16 percent over Context Release configuration when applying the nonrestrictive Alloc Release. A detail explanation will be given below.

Register Release Spill and Delayed Spill/Fill. Fig. 14 compares the performance of configurations 3 (Context Release and Register Release Spill) and 4 (Context Release, Register Release Spill, and Delayed Spill/Fill) mentioned above. The use of Delayed Spill/Fill consistently outperforms the Register Release Spill configuration, with average speedups over the baseline of six and four percent, respectively. Moreover, the number of spill/fill operations generated with Delayed Spill/Fill drops from 7.3 to 2.6 percent of the total number of committed instructions, and it almost equals the number of spill/fill operations generated for the baseline scheme, that is, 2.4 percent of the total number of committed instructions.

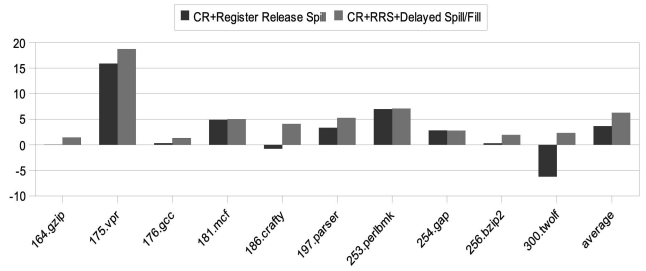

Fig. 14. Performance evaluation of Register Release Spill configuration and Delayed Spill/Fill configuration. Speedups are normalized to the baseline register window scheme with 160 physical registers.

Such a reduction produces notable performance improvements because it not only reduces memory traffic, but also the amount of physical registers allocated by fill operations. On average, by using the Delayed Spill/fill technique, fill operations drop from 4.3 to 1.3 percent of the total number of committed instructions, in comparison to the Register Release Spill configuration (spill operations generated are reduced from 2.7 to 1.4 percent). It is especially beneficial for *twolf*, where the performance improvement of Delayed Spill/Fill configuration achieves an advantage of more than eight percent over the Register Release Spill configuration. This is because by using only Register Release Spill, physical registers obtained from dirty region are required again for fill operations when the procedure returns, even though they were not necessary. Actually, it is quite often that a br.ret is followed by another br.ret, so none of the required physical registers are used. Hence, by applying Delayed Spill/Fill technique, only required fill operations (and physical registers) are performed.

A similar effect explains the performance advantage of 16 percent of *twolf* when using a nonrestrictive Alloc Release together with Context Release in comparison to Context Release, as shown in Fig. 13. This is because physical registers freed by Alloc Release come from dead values that do not need to be restored again to the map table. This also explains that a nonrestrictive Alloc Release configuration outperforms the Register Release Spill configuration by almost 4 percent for the case of *twolf*, when comparing Figs. 13 and 14 (both use the same baseline). However, the Alloc Release configuration does not adapt as well as the Delayed Spill/Fill configuration to register requirements at runtime. On average, the Delayed Spill/Fill configuration outperforms the Alloc Release configuration by almost four percent.

Such a performance improvement is mainly due to a huge reduction of the average register lifetime. Fig. 15 shows the average register lifetime of the set of the studied integer benchmarks, when using different configurations: Delayed Spill/Fill, Alloc Release, Context Release, and the register window baseline scheme. All configurations execute in a processor with a 128 register file size. Our best proposal, i.e., the Delayed Spill/Fill configuration, reduces the lifetime by 75 percent over the register window baseline scheme, and by 31 and 37 percent over the Alloc Release and the Context Release, respectively. Hence, by reducing the lifetime, the rename stage stalls caused by a lack of physical registers are also drastically reduced, increasing considerably the processor throughput.

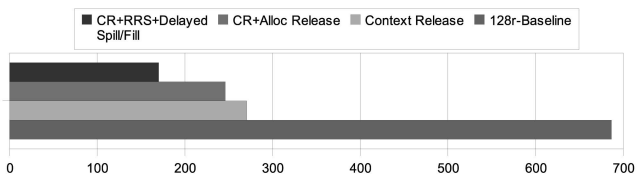

Fig. 15. Average lifetime of the set of integer benchmarks executing in a processor with a 128 register file size, when applying different configurations: Delayed Spill/Fill, Alloc Release, Context Release, and register window baseline.

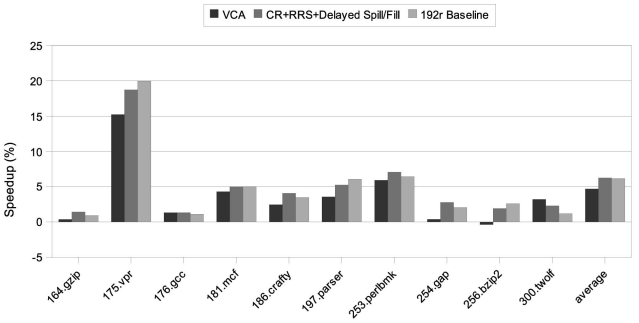

Fig. 16. Performance evaluation of VCA scheme, our Delayed Spill/Fill configuration, and the baseline register window scheme with 192 physical registers. Speedups normalized to the baseline register window scheme with 160 physical registers.

4.4 Performance Comparison with Other Schemes

Fig. 16 compares the performance of our best proposal (configuration 4) that includes Context Release, Register Release Spill, and Delayed Spill/Fill, to the VCA register window scheme. Our scheme outperforms the VCA on all benchmarks, except in *twolf*, where it loses by one percent. On average, our scheme achieves a performance advantage of two percent over the VCA. The graph also shows for comparison the performance of the baseline scheme but giving it the advantage of a large 192 physical register file (labeled as 192r Baseline), and assuming no increase on its access latency. On average, our 128-registers scheme achieves the same performance as the optimistic 192-registers baseline scheme.

4.5 Performance Impact of the Physical Register File Size

This section provides two performance comparisons of the Delayed Spill/Fill configuration 4 and the baseline scheme, for a wide range of 129-256 physical registers and 96-256 physical registers, respectively. Two different sets of binaries have been used. The first set has been compiled limiting the size of the register window up to 96 architectural registers (as it is defined in the IA-64 ISA). These binaries have been simulated for various physical register file sizes ranging from 128 to 256. The second set has been compiled with a maximum register window size of 64 architectural registers and simulated for physical register file sizes from 97 to 256. In both comparisons, performance speedups are normalized IPCs relative to the baseline configuration that uses the minimum required number of physical registers, i.e., 97 and 129 baseline register schemes for each set of binaries.

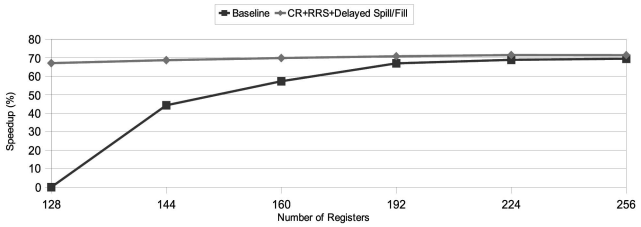

Fig. 17. Performance evaluation of our Delayed Spill/Fill configuration and the baseline register window scheme, when the number of physical registers varies from 128 to 256.

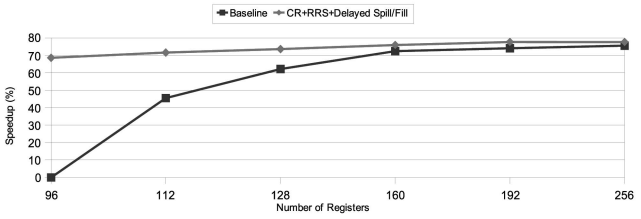

Fig. 18. Performance evaluation of our Delayed Spill/Fill configuration and the baseline register window scheme, both with register window sizes up to 64 entries, when the number of physical registers varies from 96 to 256.

Fig. 17 compares the performance when the number of physical registers varies between 128 and 256. As expected, the baseline improves the performance by increasing the number of registers, up to a saturation point around 192, beyond which it only gets marginal additional improvements (Farkas et al. [8] show similar trends when increasing the number of physical registers). As shown in the graph, our scheme consistently outperforms the baseline. However, the most remarkable result is that the baseline curve drastically degrades as the number of registers decreases (up to a 80 percent speedup), while our proposal suffers just a very small performance loss (less than four percent speedup). On average, our scheme is capable to achieve the same performance as the 192-registers baseline despite having only the minimum number of physical registers (i.e., the number of architected registers plus one).

Finally, Fig. 18 compares the performance when the number of physical registers varies between 96 and 256. Our scheme consistently outperforms the baseline. When reducing the number of physical registers from 256 to only 96, our scheme suffers only a small performance loss (seven percent slowdown), while it achieves a enormous speedup (almost 70 percent) in comparison to the baseline scheme with 96 physical registers. On average, our scheme is capable to achieve the same performance as the 160-registers baseline with only the minimum number of physical registers.

In conclusion, the Delayed Spill/Fill configuration is able to reduce the number of required physical registers by up to 64 registers, with a minimal performance loss.

5 Cost and Complexity Issues

The previous section evaluates our proposal only in terms of performance. However, it is also interesting to analyze it in terms of cost and hardware complexity.

We have proposed a low-cost implementation of the rename logic to support register windows on out-of-order processors. Compared to an in-order processor with register windows, our scheme only adds the ACDT, three map table pointers, and the *pending spill/fill* bits for each entry in the Map Table. We have experimentally found that a simple eight-entry ACDT table achieves near-optimal performance.

Compared to the VCA approach, our scheme is substantially less complex, since our proposal uses a conventional direct-mapped table, whereas the VCA requires a larger set-associative map table to hold memory address tags. Fitting the rename delay into the processor cycle time is typically a challenging problem because of its inherent complexity. Hence, adding complexity to the rename stage increases its latency, which may have implications on cycle time and power.

Moreover, the effectiveness of the register windows technique depends on the size of the architectural register file [28], so future implementations may take advantage of increasing the map table size, which emphasizes the importance of a simple scheme for scalability.

Finally, our proposal produces a lower number of spill operations than the VCA, which not only affects to the performance, but also reduces the power requirements. In comparison to the baseline, which generates a 0.5 percent of spill operations over the total number of committed instructions, the VCA generated up to three percent, whereas our scheme generates only up to 1.4 percent.

6 Summary and Conclusions

In this paper, we have proposed a software/hardware early register release mechanism for out-of-order processors that achieves a drastic reduction in the size of the physical register file leveraging register windows. This mechanism consists of two techniques: *Context Release* and *Register Release Spill*. They are based on the observation that if none of the instructions of a procedure are currently in-flight, not all mappings of the procedure context need to stay in the map table. These mappings, called *not-active*, can be released, as well as their associated physical registers.

The *Context Release* technique releases mappings defined by a procedure whose closing return instruction has committed. The values of these mappings are *dead values* that will never be used again, so their associated physical registers can be safely released.

The *Register Release Spill* technique is automatically triggered when the rename stage runs out of physical registers and releases not-active mappings that belong to caller procedures. However, unlike the previous technique, the values contained in these mappings are *live values* that may be used in the future, after returning from the procedure, so they must be spilled before releasing them.

Besides, we have developed another technique, called *Alloc Release*, which relies on the compiler to introduce alloc instructions in order to shrink the context and release all mappings that lay outside the new shrunk context and their associated physical registers. In order to explore the potential of the Alloc Release technique, we have simulated an upper bound scheme based on the knowledge of the live registers at the entry of each basic block. However, our experiments have shown that the speedup achieved by the upper bound scheme does not make up for the code size increment.

We introduce the *ACDT* that tracks all uncommitted context states to accomplish two main purposes: identify the active mappings at any point in time, which is required by our early register techniques; and implement precise state recovery of context descriptor information in case of branch mispredictions and other exceptions. We also introduce the *Retirement Map Table* that holds the state of the *Rename Map Table* at commit stage, and it allows the processor to release not-active mappings from contexts that have already been redefined at the rename stage by new contexts.

Moreover, in order to avoid unnecessary rename stalls caused by the spills and fills generated by *alloc* and *br.ret* instructions, respectively, we propose to defer spills and fills operations until the corresponding registers are used. Hence, a fill is generated and its corresponding physical register is allocated when the mapping is actually used by a subsequent instruction. Many fill operations are then eliminated, which results in a substantial reduction of the physical register file pressure. Analogously, a spill is generated when the mapping is actually reassigned by a subsequent instruction.

By applying these techniques, the average register value lifetime is reduced by a 75 percent, so a processor fitted with only the minimum required number of physical registers, i.e., the number of architected registers plus one (which is 128 in our experiments with IPF binaries), achieves almost the same performance as a baseline scheme with an unbounded register file. Moreover, in comparison to previous techniques, our proposal has much lower hardware complexity and requires minimal changes to a conventional register window scheme.

ACKNOWLEDGMENTS

This work is supported by the Spanish Ministry of Education and Science and FEDER funds of the EU under contracts TIN 2004-03072, TIN 2004-07739-C02-01, and Intel Corporation.

REFERENCES

[1] Opencc, The Open Research Compiler, http://www.open64.net/home.html, 2010.

[2] Standard Performance Evaluation Corporation. Spec., *Newsletter,* Sept. 2000.

[3] H. Akkary, R. Rajwar, and S.t. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," *Proc. 36th Ann. Int'l Symp. Microarchitecture (MICRO 36),* 2003.

[4] D. Balkan, J. Sharkey, D. Ponomarev, and A. Aggarwal, "Address-Value Decoupling for Early Register Deallocation," *Proc. Int'l Conf. Parallel Processing,* pp. 337-346, 2006.

[5] J.-L. Cruz, A. Gonzalez, M. Valero, and N.P. Topham, "Multiple-Banked Register File Architectures," *Proc. 27th Ann. Int'l Symp. Computer Architecture (ISCA '00),* pp. 316-325, 2000.

[6] D.L. Weaver and T. Germond, *SPARC Architecture Manual (Version 9),* 1994.

[7] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose, "Increasing Processor Performance through Early Register Release," *Proc. Int'l Conf. Computer Design (ICCD),* 2004.

[8] K. Farkas, N. Jouppi, and P. Chow, "Register File Considerations in Dynamically Scheduled Processors," *Proc. Second Int'l Symp. High-Performance Computer Architecture (HPCA '96),* pp. 40-51, 1996.

[9] A. Gonzalez, J. Gonzalez, and M. Valero, "Virtual-Physical Registers," *Proc. Fourth Int'l Symp. High-Performance Computer Architecture (HPCA '98),* 1998.

[10] L. Gwennap, "Intel, hp Make Epic Disclosure," *Microprocessor Report,* vol. 11, pp. 1-9, Oct. 2001.

[11] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology J. Q1,* vol. 5, no. 1, pp. 1-13, Feb. 2001.

[12] "Intel Corporation," *Intel Itanium Architecture Software Developer's Manual,* Volume 2: System Architechiture, 2002.

[13] T.M. Jones, M.F.P. O'Boyle, J. Abella, A. Gonzalez, and O. Ergin, "Compiler Directed Early Register Release," *Proc. 14th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '05),* pp. 110-122, 2005.

[14] T.M. Jones, M.F.P. O'Boyle, J. Abella, A. González, and O. Ergin, "Exploring the Limits of Early Register Release: Exploiting Compiler Analysis," *ACM Trans. Architecture and Code Optimization,* vol. 6, no. 3, pp. 1-30, 2009.

[15] A. KleinOsowski and D.J. Lilja, "Minnespec: A New Spec Benchmark Workload for Simulation-Based Computer Architecture Research," *IEEE Computer Architecture Letters,* vol. 1, no. 1, p. 7, Jan. 2002.

[16] J.L. Lo, S.S. Parekh, S.J. Eggers, H.M. Levy, and D.M. Tullsen, "Software-Directed Register Deallocation for Simultaneous Multithreaded Processors," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 9, pp. 922-933, Sept. 1999.

[17] M.M. Martin, A. Roth, and C.N. Fischer, "Exploiting Dead Value Information," *Proc. 30th Ann. Int'l Symp. Microarchitecture (MICRO 30),* 1997.

[18] J.F. Martinez, J. Renau, M.C. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed Early Resources Recycling in Out-of-Order Microprocessors," *Proc. 35th Ann. Int'l Symp. Microarchitecture (MICRO 35),* 2002.

[19] T. Monreal, A. González, M. Valero, J. González, and V. Viñals, "Delaying Physical Register Allocation through Virtual-Physical Registers," *Proc. 32nd Ann. ACM/IEEE Int'l Symp. Microarchitecture (MICRO 32),* pp. 186-192, 1999.

[20] T. Monreal, V. Viñals, A. Gonzalez, and M. Valero, "Hardware Schemes for Early Register Release," *Proc. Int'l Conf. Parallel Processing (ICPP-02),* 2002.

[21] M. Moudgill, K. Pingali, and S. Vassiliadis, "Register Renaming and Dynamic Speculation: An Alternative Approach," *Proc. 26th Ann. Int'l Symp. Microarchitecture (MICRO 26),* pp. 202-213, Nov. 1993.

[22] D.W. Oehmke, N.L. Binkert, T. Mudge, and S.K. Reinhardt, "How to Fake 1000 Registers," *Proc. 38th Ann. Int'l Symp. Microarchitecture (MICRO 38),* pp. 7-18, 2005.

[23] I. Park, M.D. Powell, and T.N. Vijaykumar, "Reducing Register Ports for Higher Speed and Lower Energy," *Proc. 35th Ann. ACM/IEEE Int'l Symp. Microarchitecture (MICRO 35),* pp. 171-182, 2002.

[24] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation," *Proc. 37th Ann. Int'l Symp. Microarchitecture (MICRO 37),* pp. 81-92, 2004.

[25] E. Quinones, J.-M. Parcerisa, and A. Gonzalez, "Selective Predicate Prediction for Out-of-Order Processors," *Proc. 20th Ann. Int'l Conf. Supercomputing (ICS '06),* 2006.

[26] E. Quinones, J.-M. Parcerisa, and A. Gonzalez, "Early Register Release for Out-of-Order Processors with Register Windows," *Proc. 16th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '07),* 2007.

[27] E. Quinones, J.-M. Parcerisa, and A. Gonzalez, "Improving Branch Prediction and Predicate Execution in Out-of-Order Processors," *Proc. 13th Int'l Symp. High-Performance Computer Architecture (HPCA '07),* Feb. 2007.

[28] R. Rakvic, E. Grochowski, B. Black, M. Annavaram, T. Diep, and J.P. Shen, "Performance Advantage of the Register Stack in Intel Itanium Processors," *Proc. Workshop Explicit Parallel Instruction Computing (EPIC) Architectures and Compiler Technology,* 2002.

[29] M. Vachharajani, N. Vachharajani, D.A. Penry, J.A. Blome, and D.I. August, "Microarchitectural Exploration with Liberty," *Proc. 35th Ann. Int'l Symp. Microarchitecture (MICRO 35),* pp. 271-282, 2002.

[30] R. Yung and N. Wilhelm, "Caching Processor General Design," *Proc. Int'l Conf. Computer Design (ICCD),* pp. 307-312, Oct. 1995.

[31] V. Zyuban and P. Kogge, "The Energy Complexity of Register Files," *Proc. Int'l Symp. Low Power Electronics and Design,* pp. 305-310, 1998.

Eduardo Quiñones received the MS and PhD degrees from the Universitat Politecnica de Catalunya in 2003 and 2008, respectively. He is a senior researcher at BSC. His area of expertise is in safety-critical systems and high-performance compiler techniques. He is involved in several FP7 projects (MERASA, PROARTIS). He is a member of the IEEE.

Joan-Manuel Parcerisa received the MS and PhD degrees in computer science from the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, in 1993 and 2004, respectively. Since 1994, he has been a full-time assistant professor in the Computer Architecture Department, Universitat Politècnica de Catalunya. His current research topics include clustered microarchitectures, multithreading, and acceleration of emerging applications.

Antonio González received the MS and PhD degrees from the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain. He is the founding director of the Intel-UPC Barcelona Research Center, started in 2002, whose research focuses on new microarchitecture paradigms and code generation techniques for future microprocessors. Prior to his work at Intel, he joined the faculty of the Computer Architecture Department of UPC in 1986 and became a full professor in 2002. He currently holds a part-time full professor position at this department. He has published more than 300 papers, has given more than 80 invited talks, has filed more than 40 patents, and has advised 18 PhD thesis in the areas of computer architecture and compilers. He has served as an associate editor of the *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems, ACM Transactions on Architecture and Code Optimization*, *IEEE Computer Architecture Letters*, and *Journal of Embedded Computing*. He has served on the program committees for more than 100 international symposia in the field of computer architecture, including ISCA, MICRO, HPCA, ASPLOS, PACT, ICS, ISPASS, CASES, and IPDPS. He has been the program chair for ICS 2003, ISPASS 2003, MICRO 2004, and HPCA 2008, and the general chair for MICRO 2008 among other symposia. His awards include the award to the best student in computer engineering in Spain among those graduating in 1986, the 2001 Rosina Ribalta Award as the advisor of the best PhD project in Information Technology and Communications, and the 2008 Duran Farrell Award for research in technology. He is a member of the IEEE and the IEEE Computer Society.